Design and Implementation of a Distributed Content-based Notification Broker for WS-Notification

Andres Quiroz and Manish Parashar
TASSL, Department of Electrical and Computer Engineering, Rutgers University
94 Brett Road Piscataway, NJ 08854
{aquirozh, parashar}@caip.rutgers.edu

Abstract— **We describe an implementation based on the WS-Notification (WSN) specification for Publish/Subscribe communication which provides a distributed, content-based notification service. The implementation is based on a distributed hashtable (DHT) built on a structured overlay of peer nodes. The entire system acts as a notification broker, so that notification producers and consumers that make use of the network can achieve loosely-coupled communication with a decentralized, scalable service. We develop and evaluate self-optimizing behavior built to reduce notification traffic within the network.**

I. Introduction

Web services have emerged as one of the key enabling technologies for Grid systems, providing platform-independent interactions between distributed applications and resources. The WS-Notification (WSN) specification [1] is a set of web service standards that define protocols to realize the publish/subscribe communication pattern. WSN was developed within the context of the WS-Resource Framework (WSRF), which describes how to implement OGSA capabilities using Web services [2]. Thus, it is expected that Grid applications will adopt the WSN protocols for their communication infrastructure.

To date, however, there have been few implementations of the WSN specifications, as it is still an emerging standard. Most of the existing implementations are bindings to specific languages that provide the basic functionality for WSN objects and message exchanges, and include API's that developers can extend for specific applications. For example, Apache's Pubscribe [3] is such an implementation for Java, while WSRF.NET from the University of Virginia's Grid Computing Group [4] implements the standard for Microsoft's development platform. However, these implementations are purposefully generic, and do not address some of the open issues not regulated by the standard, such as mechanisms for the location of producers and/or consumers, and the efficient and scalable management of subscriptions and routing of notifications. While it is clearly evident why the standards cannot seek to regulate these application dependent issues, it should also be noted that these issues are common enough to warrant the implementation of services that provide these mechanisms for application development.

In fact, the means for providing such services is anticipated in WSN by the definition of the WS-BrokeredNotification (WSBrN) specification. A notification broker is a service that mediates interactions between producers and consumers, and that is expected to relieve them of specific tasks. A broker may be a well-known rendezvous point through which producers and consumers can find each other and exchange messages. However, which specific tasks are assumed by the broker and how they must be implemented are issues that should not and are not regulated by the brokered notification standard. Furthermore, they are not trivial in distributed environments such as the Grid, where scalability and dynamism are prevalent challenges.

This paper describes an implementation of a notification broker service for subscription management and notification dissemination targetting highly dynamic pervasive Grid environments that adopt the WSN standards. The service is based on a distributed and decentralized architecture, which uses a content-based indexing scheme and rendezvous-based communication model to realize the different WSN operations. The system is designed as an associative Distributed Hashtable and its implementation builds on the Meteor communication middleware [5]. To further support the efficiency and scalability of our approach, we design self-optimization mechanisms for reducing the number of messages transmitted by the system. These optimizations are meant to alleviate the overhead of notification flows. We show experimental data that supports the effectiveness of the self-optimizing behavior.

The rest of the paper is organized as follows. Section II presents a brief overview of the WSN family of specifications and identifies the challenges for their implementation particular to Grid environments. Section III then presents our system design and functionality. Section IV provides the details of our implementation, followed by Section V, which describes and evaluates the system's self-optimizing mechanisms. We outline related work in Section VI and finally present our conclusions and directions for further work in Section VII.

[0]The research presented in this paper is supported in part by National Science Foundation via grants numbers ACI 9984357, EIA 0103674, EIA 0120934, ANI 0335244, CNS 0305495, CNS 0426354 and IIS 0430826, and by Department of Energy via the grant number DE-FG02-06ER54857.

II. Overview of WSN

The WSN specification consists of 3 interrelated standards: WS-BaseNotification (WSBN) [6], WS-BrokeredNotification (WSBrN) [7], and WS-Topics (WST) [8]. WSBN specifies the basic elements of the notification pattern: the NotificationConsumer (NC) that accepts notification messages, the NotificationProducer (NP) that formats and generates these messages, and the Subscription, a consumer-side initiated relation between a producer and a consumer. The only fixed field in a subscription is the consumer reference, which by itself implies the consumer's interest in all of the notifications generated by the producer to which the subscription was made. Optionally, a subscription can contain a filter, specified as a FilterType element, that forces a producer to send only those notifications that pass (match) the filter. WSBN does not regulate the syntax or use of the FilterType element, but suggests three basic types: topic filters, message content filters, and producer properties filters. WSBN also regulates subscription management, which the consumer can perform given the reference it receives in response to a subscription. This reference is meant to contain enough information to enable it to contact and interact directly with the subscription as a resource, as defined by WSRF. Thus, subscription operations (unsubscribe, renew, pause, and resume) do not include a subscription reference as a parameter. In addition to the push-style pattern of notification, where producers send notifications directly to consumers, WSBN defines a pull-style pattern, where messages are stored at a pre-defined location (a pull-point) until they are retrieved by the consumer.

WSBrN defines the NotificationBroker (NB) entity and its expected functionality. A notification broker is an intermediary Web service that decouples NC's from NP's [7]. A broker is capable of subscribing to notifications on behalf of consumers and is capable of disseminating notifications on behalf of producers. Consumers and producers thus interact dynamically and anonymously through the NB without the need for explicit knowledge of each other's identities or locations. Management of this knowledge is delegated to the broker. A NB essentially implements the NC, NP, and other interfaces defined in WSBN. As a specific functionality, a notification broker can accept producer registrations, which is meant for realizing the demand-based publishing pattern. Using this pattern, publishers avoid the (possibly expensive) task of creating notifications when no subscriptions (and, thus, no consumers) exist for them. To this end, a NP must register with the NB, providing a set of topics. When subscriptions are made that correspond to or include topics in a particular producer's registration, the NB subscribes to the producer for those topics. Only then does the producer start sending notifications.

Finally, WST tries to standardize the way in which topics are defined, related, and expressed. It defines the notion of a topic space, where all of the topics for an application domain should be defined and organized, possibly in a hierarchical way. A topic expression is the representation of a particular topic or range of topics. The syntax of a topic expression is identified by the topic expression's dialect. WST defines three dialects: Simple, Concrete, and Full. A simple topic expression is just the qualified topic name. A concrete topic expression is used for hierarchical topic spaces, and is given in a path notation, such as in `myNamespace:news/tv/cnn`. Here `myNamespace` identifies the topic space, and each of the subsequent identifiers belong to successively deeper levels in the hierarchy. A full topic expression is the same as a concrete expression, except that it uses special operators and wildcard sequences for spanning multiple topics within the topic hierarchy.

A. *Implementation Issues in Grid Environments*

The nature of pervasive Grid environments poses specific challenges to the implementation of WSBrN that must be reflected in its design. Grids are large-scale systems where many potential producers and consumers (sensors, computation services, agents, etc.) need to exchange information. To handle the large number of potential clients, the notification service must be distributed, decentralized, and scalable. Service providers participate by making nodes available to the notification system, and can in turn make use of the system through these nodes. A peer-to-peer design avoids the need for centralized control and gives the service providers the flexibility to join or leave the system at will. This behavior requires a dynamic and self-configuring underlying infrastructure.

Interactions between Grid components are also dynamic and highly decoupled. The dynamicity in the types of interactions possible in a pervasive Grid implies that they cannot be predefined and classified, as is the case with topic spaces in WSN. Furthermore, interactions may be made across application or organizational domains so that global synchronization of naming conventions is not feasible. Content-based interactions [9] can overcome these limitations by being based on a global information space from which notification content is drawn, which is synchronized on a semantic level. Thus, depending on the mechanisms used, interactions can potentially be as varied and dynamic as notifications themselves.

III. Distributed Content-based Notification Broker

Our system, referred to henceforth as the notification service, is a distributed and decentralized notification broker. Each of the nodes within the system is a peer that implements the NotificationBroker interface, and an external client (producer or consumer) can interact with any of them. Thus, the whole system acts as a single NB, as illustrated in Figure 1. This is important because the interface is not a bottleneck, and the system has no single point of failure.

Through this interface, clients and brokers realize the message exchanges defined in the WSN specifications, using XML messages that represent subscriptions, notifications, etc. Topic expressions are used by the notification service as identifiers for these messages, and provide the means for matching between them. Unlike a purely topic-based system, such as

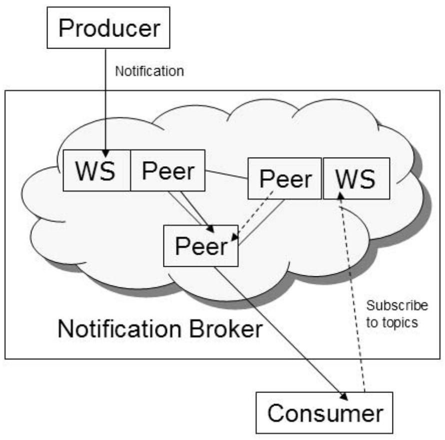

Fig. 1. Layout of the broker network. Matching subscriptions and notifications will be routed to the same rendezvous node, which will perform the matching and relay the notification.

WSN with topic filtering, topics in the notification service are meant to be content-based. The idea is to remain as close as possible to the notation and semantics defined in WS-Topics for topic expressions, both to support applications that implement this standard and to simplify content-based indexing, avoiding having to parse the payload content (which is necessary when using message content filtering as described in WSN).

Topics in the notification service are expressed using the same notation as in the Concrete dialect of WST (see Section II); the difference is that, while normally the topic is an atomic unit that should correspond to an actual path in the topic space definition, here each of the identifiers is taken as a value from a separate dimension in a multi-dimensional information space, where only the type of each dimension is defined. Thus, the range of topics is limited only by the possible combinations of the values along each dimension.

To observe the difference, consider a weather monitoring application that subscribes to sensor data. In this example, the application may define the information space with three dimensions: geographic area, barometric pressure, and temperature. A topic for a notification in this system might then be `weatherService:Piscataway,NJ/29/48.9`, while a subscription could be `weatherService:*,NJ/>25/<50`. Defining a hierarchical topic space for this type of topic expression would not be practical, since individual topic identifiers would be needed for each geographic location, and, worse still, for each numeric value.

The system uses a rendezvous-based messaging model [10], in which matching messages "meet" at some node within the network, referred to as a rendezvous node (see Figure 1). The messaging model also applies the concept of reactive behaviors, by which the behaviors at rendezvous nodes are determined by actions embedded in the message request (in this case, subscribe, notify, etc.).

A Distributed Hashtable (DHT) is a data structure that associates objects identified with a set of keys with nodes in a distributed system and that provides the mechanisms to store or retrieve these objects through a put/get interface. In our system design, a DHT is used to map matching topics to the same nodes, where the specified actions are carried out. This way, each node manages a distinct subset of topics that map to it. However, the mapping used by the DHT must handle the possibility of complex topic expressions used to span multiple simple topics through the use of wildcards and ranges. This cannot be done using simple hashing. Instead, the multidimensional information space used by the content-based indexing scheme is mapped to a 1-dimensional identifier space for physical nodes in the network. For any single topic, the identifier that it maps to is assigned to the node in the network with the closest succeeding identifier. This mechanism is able to map complex topics obtained by wildcards and ranges to "clusters" of consecutive identifiers that correspond to the identifiers of the individual topics. This guarantees that complex topics and simple topics will meet as required and reduces the number of nodes that correspond to complex topics (because consecutive identifiers are likely to be assigned to a single node).

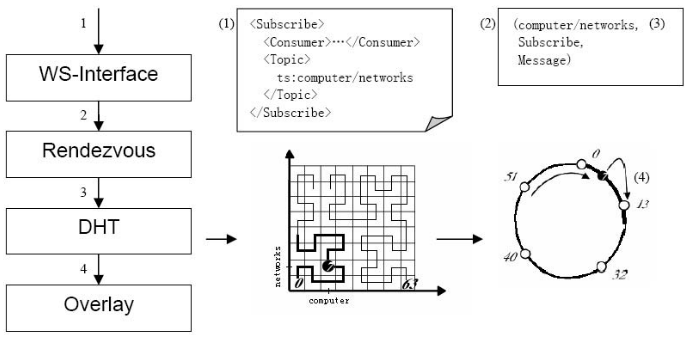
```
<Subscribe>
  <Consumer>…</Consumer>
  <Topic>
    ts:computer/networks
  </Topic>
</Subscribe>
```
```
(computer/networks,
Subscribe,
Message)
```

Fig. 2. Architecture and behavior of the distributed system. An XML message is received by the WS interface, and the action and topic expression are extracted. The topic expression is mapped by the DHT from its multidimesional space to a node ID, which it then uses to route the message on the overlay to the rendezvous node, which must execute the corresponding action.

Finally, the nodes that make up the peer network are organized as a structured overlay network, which guarantees bounded costs in terms of number of messages and number of nodes involved in routing. The overlay also has self-managing capabilities for reconfiguring in the face of the dynamic arrival and departure of nodes. Figure 2 illustrates the basic functionality of the architecture.

A. Subscribe and Notify

As a NP, the broker accepts subscriptions from clients. Subscriptions handled by the notification service must contain a topic expression within the FilterType element, as explained in the previous section. When a subscription message is received by any one of the brokers, its topic expression is decomposed into its constituent values and mapped by the DHT to the node identifier space. Since a subscription topic can contain wildcards or ranges, the subscription may span multiple topics, which may correspond to one or several nodes.

These nodes store the subscription until the termination time that is part of the subscription message. If no termination time is specified, then the subscription is kept indefinitely or until cancelled by the client (a system-wide maximum TTL can be employed to avoid keeping old subscriptions that are never cancelled). To produce a unique identifier for the subscription, the entire topic is hashed, along with the consumer endpoint reference. This ensures the differentiation of subscriptions for the same topic or topics from different consumers. The unique identifier is appended to the topic expression, which is returned to the client as the subscription reference to be used for subscription management (as described in the next section).

Notifications are handled by brokers, acting as NC's, upon invocation of the Notify method. The procedure is similar to that of a subscription. If the notification's topic expression is singular, in the sense that it does not contain wildcards nor ranges that span multiple topics, then the notification maps to a single rendezvous node within the network. If a subscription for that topic exists at the rendezvous node, then the consumer reference is extracted from the subscription record stored at the node and used to connect to the client and relay the message. Figure 1 also illustrates this rendezvous process. If a notification is identified by a topic expression that spans mutliple topics, the notification isn't routed to a rendezvous node as above. The reason for this is that there might exist multiple rendezvous nodes for it, a number of which may store the same matching subscription, resulting in the same notification being relayed to a consumer multiple times. Instead, the interface node that received the notification queries the network for subscriptions, and then directly relays the notification only to each different consumer reference from the subscriptions it receives. A similar procedure is also used for demand-based publishing, as explained in Section III-D.

B. Subscription Management

As was mentioned in the previous section, a subscription reference in the notification service consists of a topic expression and a unique identifier. The functions defined by WSN for the SubscriptionManager and PausableSubscriptionManager interfaces depend on this subscription reference because, given that nodes can enter and leave the system at any time, subscriptions cannot be tied to a particular rendezvous node. Recall from Section II that in WSRF an endpoint reference can be used to interact with a subscription resource directly. However, because of the above and the fact that we do not assume that subscriptions are WSRF resources, such endpoint references are not used. Thus, the topic expression is always used to route the requests to the node(s) at which the corresponding subscription is currently stored. Once at these nodes, the subscription's unique identifier is used to quickly obtain the particular subscription and execute the appropriate action.

C. Pull-Style Notification

Pull-style notification in the notification service is done in a very similar way to the way regular subscriptions are handled. The only difference is that, when a pull-point creation request is made to a broker, a message repository is also created at each rendezvous node where the subscription is stored. Notifications are stored in these repositories rather that being relayed directly to the consumers. Finally, when a consumer invokes the `GetMessages` command on a broker, it queries the network with the subscription reference to obtain the notifications stored at the repositories, constructs a single response with all of these notifications, and sends them back to the client.

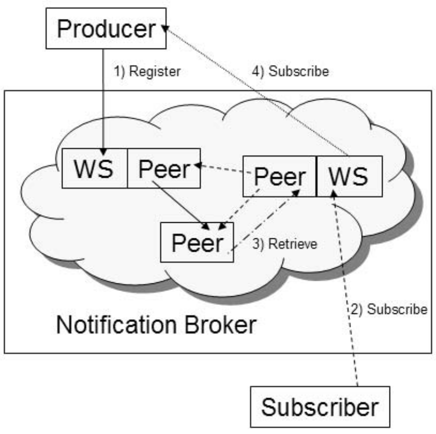

Fig. 3. Publisher registration and subsequent subscription. Notice that a subscription and notification are not necessarily the same, but as long as they overlap as some node, the registration will be retrieved.

D. Demand-based Publishing

Publisher registration occurs in the notification service in exactly the same way as a subscription. The registration topic is used to route a registration message to a node or nodes in the network. In order to accommodate demand-based publishing, however, the procedure for a subscription detailed in Section III-A must now include a query for publisher registrations that match (or, rather, overlap with) the subscription's topic expression. If such registrations exist, then the NB that received the original subscription subscribes in turn to the producer(s) for the topic(s) given in the registration(s). Figure 3 illustrates this mechanism.

IV. Implementation Details

A. WSBrN Service

The Web Service interface for the notification broker was implemented in Java using the JWSDP 2.0 API and development tools. First, the XML schema for base notification and brokered notification, provided in [1], were transformed into Java objects using the JAXB binding tools. There are a number of places in the schema where the element type `xsd:any` is used to indicate configurable parameters for the message elements. In order to conform to both the Java platform and our own implementation architecture, some of these elements were redefined:

- The FilterType element from the WSBN schema was redefined to contain a topic expression because topics are used for defining subscriptions.

- The TopicExpressionType element was redefined to contain a `xsd:string` element for holding topic strings as defined in WST.
- The Message element in the NotificationMessageHolderType definition was redefined to be of type `xsd:string`. This is not a limitation, since the message is application specific data that can be encoded as and interpreted from a string. This is mainly to facilitate its manipulation in Java.
- A SubscriptionReference element was added to the messages defined for subscription management (Renew, Unsubscribe, etc.), which is required to find particular subscriptions (Section III-B).

After the Java objects were created from the schema, the WSDL documents were used to create the Java service interfaces and implementing classes, which were then deployed as a service endpoint for a NB. This endpoint can then be run on an Apache or similar web service container to receive and respond to client requests.

Earlier, we mentioned that every network node in principle implements the NotificationBroker interface. However, as can be noted in Figures 1 and 3, not all nodes need do so. This is important because some nodes may not have access to or be able to run a web service platform. Thus, only nodes with such capability will serve as the interface with the system; some of these may be well-known service endpoints or may be registered in a directory service for discovery. Other system nodes may implement the WS interface or run only as DHT nodes and provide and perform subscription management, matching, and notification tasks.

B. Underlying Infrastructure

We based our implementation on the Meteor content-based communication middleware [5]. Meteor supports the paradigm of Associative Rendezvous (AR), which is similar to the design presented here, and also applies the concept of reactive behaviors, which allows the multiple interaction semantics defined in WSN to be implemented. DHT functionality in Meteor is provided by a content-based mapping and routing infrastructure known as Squid [11]. Squid uses Hilbert space filling curves to realize the content-based mapping as described in Section III and optimizes message routing based on this mapping. The structured overlay that supports the peer network is the Chord [12] overlay network, which is a one-dimensional overlay as employed by the mapping mechanism, and effectively implemententes the self-managing behavior required for handling the dynamic arrival and departure of nodes. This overlay also provides the system with a level of fault tolerance, given that routing capabilities can be preserved in the face of node failures. These system layers are each implemented as JXTA services, so that peer discovery and data message transport are carried out by the JXTA framework.

The Meteor and Squid systems have been deployed on various distributed platforms, including a university-wide Grid and the PlanetLab [13] planetary-scale distributed testbed, as well through simulations. The performance of these systems has been evaluated with respect to end-to-end latencies, numbers of nodes involved in queries and routing, and load distribution and balancing. For results of these evaluations, which demonstrate the scalability of these systems, please refer to the following publications [11], [5], [14].

V. Self-optimization Mechanisms

The number of messages sent within the system can be reduced at the notifications level, which is important because any reduction in the number of messages leads to a reduction in the overhead involved in packaging and delivering each individual message, and to an improvement in scalability. In the case of Web Services, this overhead is incurred mainly by XML and SOAP headers. In addition, the messaging within the JXTA framework also adds considerable overhead. To see how much bandwidth is actually consumed by overhead in one implementation, a sniffer program was used to capture the packet flows between the nodes in the network for notifications. For messages between network nodes, the combined overhead of XML and JXTA for each message is just over 3.5 KB, which amounts to about 28 Kbps in a message flow of one message per second. The following mechanisms are used by the system to reduce the number of individual messages in the network.

A. Grouping of Notifications by Buffering

This optimization is meant to reduce the flows of small and frequent notifications. A simple way to deal with these notification flows is to buffer and group several notifications within a single notification message, a mechanism which is allowed by the WSN XML schema. This way, the headers that would have been transmitted with every individual message are reduced to a single header on a grouped message. Determining when and how many messages to buffer, however, depends on several factors, and thus it is worthwhile to equip the system with logic that allows it to autonomously determine the most appropriate level of message aggregation based on high-level constraints.

Without application specific considerations, messages can be grouped based on two criteria. The first is on messages that correspond to the same topic, and the second is on messages that match the same subscription. These criteria are not necessarily the same, since, depending on how broad a subscription is made (with wildcards or ranges), several different topics will match a single subscription. The system can benefit from applying both criteria, since grouping based on topic equality can be done when messages enter the system at an interface node, which doesn't necessarily know about subscriptions for that topic, and then subscription-based grouping can be determined at the rendezvous nodes. The mechanism, however, is the same in both cases, so we will describe grouping based on topic equality.

The mechanism for grouping and packaging of notifications is as follows. Each interface node keeps a separate buffer of messages for every topic it receives (garbage collection can be employed to eliminate buffers for which no messages arrive for a period of time). Each buffer is configurable by setting the length of the period during which messages are accumulated. This buffering level is determined by managers associated with each buffer, the design of which is the real issue of this setup.

If the buffering period is determined only with respect to bandwidth utilization (the number of messages), then the solution is trivial because a higher buffering level (more messages grouped together) always increases the saving achieved. If a limit is set on the buffering period, according to the maximum latency allowed for each individual message, then the solution would always be set to this limit. However, a more balanced solution should consider the tradeoff between bandwidth utilization and message latency. An optimal point can be found between a buffering period of zero (minimum latency, maximal bandwidth consumption) and one equal to the maximum allowed latency (highest buffering level, minimal bandwidth consumption). This is the range used in Equation (1) below, although the reciprocal of the incoming rate is used as a lower bound instead of zero (any period set smaller than the incoming message period would result in no buffering). Instead of manually assigning a weight to each extreme, a dynamic solution is determined based on the relative size of the payload with respect to the total message size (Equation 2 below). The rationale behind this is that the relative saving in bandwidth is greater for small messages because the overhead constitutes a larger fraction of the total data sent, whereas for large messages the overhead becomes relatively insignificant. In the former case, there is greater payoff for sacrificing latency, and thus buffering should have a larger weight. For the latter case, the reverse is true. Finally, the period is calculated by obtaining a value within the range determined by the weight, using Equation 3. If the incoming rate is very low, with a period higher than the maximum latency, then Equation 3 is not used, and rather the period is set directly to zero.

$$\begin{aligned} range &= maxLatency - avgIncomingRate^{-1} && (1) \\ weight &= \frac{avgPayload}{overhead + avgPayload} && (2) \\ period &= maxLatency - weight \times range && (3) \end{aligned}$$

As a proof-of-concept, experiments were conducted for single message flows of different incoming rates and payload sizes. The maximum latency allowed for messages at each node is 1 second. The buffering period, as well as the actual groups of messages transmitted by the system, were observed to obtain the overhead bandwidth consumption. Figure 4 plots the results. In both graphs, the top and bottom lines correspond to the maximum and minimum values that the corresponding measure can take, given the incoming rate. The lines in between correspond to flows with payload sizes of 10, 100, 500, 1000, 5000, and 10000 bytes. In the top graph, they appear in this order from bottom to top, the overhead being lower for smaller payloads. In the bottom graph, they appear in order from top to bottom. Notice that savings in bandwidth utilization are substantial, even though buffering periods are

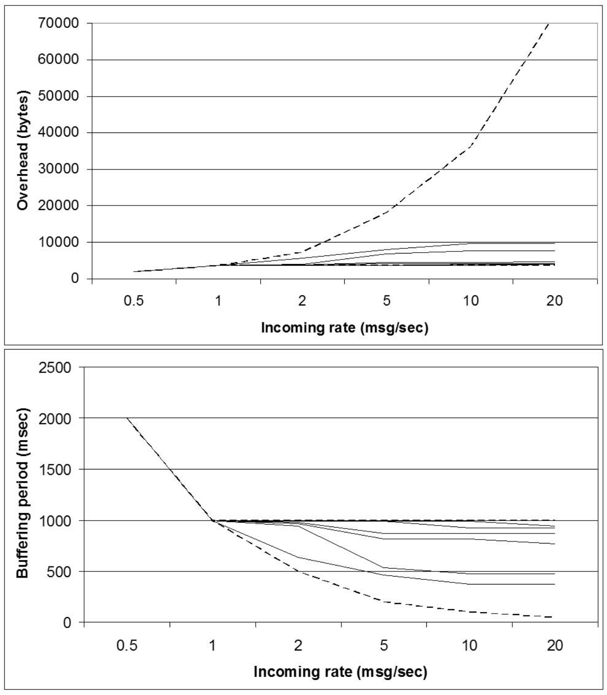

Fig. 4. Results for buffering with different incoming rates and payload sizes. Lines correspond to the different payload sizes, bracketed by the minimum and maximum values for each measure. Top: Overhead bandwidth, grows with payload size. Bottom: Buffering period set, decreases with payload size.

distributed within the range of allowable latencies. The lowest buffering period set in this case is 373 seconds for message rate of 20 messages per second and 10000 bytes per message.

For irregular notification flows, possibly originating from several producers publishing notifications on the same topic at different time intervals, several complications are possible, such as short bursts of notifications at high rates, high variability in the incoming rate, and concurrency. A number of mechanisms were used to reduce the sensitivity of the system to these conditions. To emphasize the self-managing aspect of the system, the use of fixed low level parameters was avoided. For example, instead of using a fixed threshold for the minimum change in the buffering period, the threshold is calculated dynamically based on whether or not the change in buffering would cause at least one message more or less to be buffered at the current estimated incoming message rate. This new parameter (the change in the number of messages for which a change in period is allowed) is at a higher level and is more meaningful than the period alone.

To test the behavior of the system under these conditions, an interface node was set to receive messages with the same topic from 32 different producers, each one of which sent messages of random payload size between 10 and 500 bytes at random intervals of up to 5 seconds. The combined effect of these notifications produces a high message rate, with high variability. Figure 5 shows the changes in the buffering period during the time of the test.

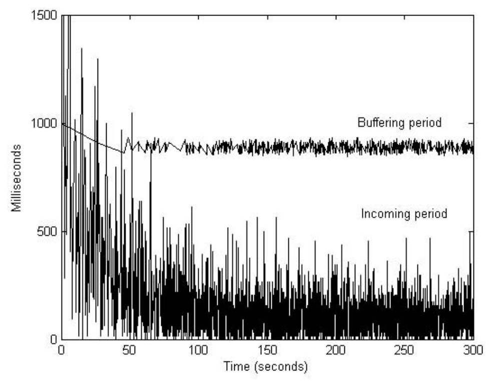

Fig. 5. Change in the buffering period with highly variable incoming periods

B. Demand-based Notification Relay

Ideally, notifications should not be sent if no subscribers exist for them. Demand-based publishing, explained in Section III-D, is WSN's provision for dealing with this issue. However, demand-based publishing depends on producers registering their topics with the notification broker, which particular publishers may not choose to do or may not be able to do if they do not implement the NP interface. To further optimize messaging, the system implements a mechanism which is similar to that of demand-based publishing but that is based on the topics of individual notifications. The idea is that interface nodes should determine when not to relay notifications to rendezvous nodes based on the existing subscriptions.

Unlike publisher registrations that define the topics that will be produced beforehand, an interface node has no way of knowing which topics it will have to handle. Registering for every topic received would also be inefficient. Thus, the mechanism devised is implemented as follows. Each interface node keeps subscription caches associated with particular topics. If there is no cache associated to a particular topic when a notification for it is received, the interface node queries the network for subscriptions for that topic. If any are found, they are placed in the subscription cache, which is marked as empty otherwise. Subsequent notifications with the same topic will only be relayed if the corresponding subscription cache is not empty. Cached subscriptions retain the TTL information, so that they will expire in the cache if they expire at the rendezvous nodes. To avoid making a query for every topic received, locality is exploited by checking a topic against all cached subscriptions. New queries are only made if no subscriptions exist in these caches (note that for these topics, the notification is relayed in any case).

Meanwhile, at the rendezvous nodes that responded to the query, a temporary registration is kept of the interface nodes and their corresponding queries. This ensures that if a subscription did not exist at the time of the query, a matching subscription made thereafter can be made known to the interested nodes, so that a notification for which a subscription exists is never dropped. The same happens for the cancellation of subscriptions. Because they are potentially more numerous than publisher registrations, rather than keeping these registrations indefinitely, they are deleted once they are used. Thus, interface nodes must requery the network once a cache for a particular topic becomes empty.

The overhead of this mechanism for each topic are the query and its corresponding response (2 messages), as well as one message per update of a subscription or its cancellation. New queries are only triggered after cancellations. Thus, the overhead is small and can easily be made up when large notification flows are not relayed while no subscriptions exist, unless rates of subscriptions and cancellations are in the same order as the rate of notifications.

VI. Related Work

In Section I, we mentioned Apache's Pubscribe [3] and WSRF.NET [4] as implementations of the WSN standard. Other implementaitons include pyGridWare [15], a Python-based implementation, the GT4 Globus Toolkit [16] with bindings for both Java and C, and WS-Messenger from Indiana University [17]. Note that Apache's Pubscribe is based on GT4-Java. The primary focus of these implementations is WSRF, and, as a result, they provide different levels of functionality for WSN. For example, the pyGridWare and GT4 implementations of WSN are meant primarily for providing notifications about the state of resource properties, although GT4 does provide comprehensive implementations of WS-BaseNotification and WS-Topics, but not of WS-BrokeredNotification. Similarly, Pubscribe fully supports WSBN and WST, but not WSBrN. One important aspect of the Pubscribe implementation is that it also fully supports WS-Eventing (WSE) [18], another publish/subscribe standard for Web services driven primarily by Microsoft and IBM, providing interoperability between the standards at a high level. WSRF.NET, which was developed using ASP.NET and the IIS infrastructure, supports all of the specifications. A thorough comparison of these implementations can be found in [19].

The above implementations are meant to be development tools, providing technology-specific bindings for the standards and extensible API's. Like the standards themselves, they do not address the issues that arise when actually composing systems that make use of the notification protocols and standards, such as service discovery, and efficient and scalable routing of requests and messages. WS-Messenger is a recent implementation of all the WSN specifications, as well as of WSE. Its main feature is its leveraging of a generic messaging interface that can be adapted to work on top of existing messaging systems that address these issues in some way. One of these systems is the NaradaBrokering middleware framework [20]. NaradaBrokering also provides its own implementation of WSE, and an implementation for WSN is currently being developed for it. The objectives and approach of the NaradaBrokering framework are essentially the same as those that underlie the work presented here; it manages a network of brokers through which end systems can interact, providing scalability, location independence, and efficient routing. The difference is that Narada brokers are organized in a hierarchical structure which must be maintained through tighter coupling and control mechanisms imposed on the participation (connection, disconnection) of broker nodes.

Content-based publish/subscribe over DHT's is a topic for which there is much current work. DHT functionality is usually built using some sort of a structured overlay network, the most popular of which are Chord [12], used here, Pastry [21], and CAN [22], because they provide scalability, search guarantees and bounds on messaging within the network, as well as some degree of self-management and fault tolerance with respect to the addition/removal of nodes. With this foundation, designing content-based publish/subscribe systems requires an efficient mapping between content descriptors and nodes in the overlay network, as well as efficient techniques for routing and matching based on these content descriptors, which can contain wildcards and ranges for complex queries. The work in [23], [24], [25], [26] addresses these issues to some extent. Meteor and Squid differ from these approaches mainly in the locality-preserving mapping used. The Meghdoot system [26] also uses a locality-preserving mapping, but the multidimensional address space used by its overlay network, CAN, is costlier to maintain than Chord's one dimensional overlay. Of these systems, [23] also proposes buffering of messages, but it does so statically by setting the buffering level as a multiple of the incoming period. To our knowledge, none of these systems are as yet used to implement the WSN standards.

VII. Conclusion

We have described the implementation of a distributed content-based notification broker service for WS-Notification in the context of large-scale, dynamic Grid environments. The issues of scalability and dynamism are addressed by our system design and by our implementation, which is based on a scalable and self-managing underlying infrastructure. We also described and evaluated self-optimization mechanisms to reduce the number of notification messages transmitted within the network.

This system is yet to be tested in a WAN setting with real application data. As described in [19], compatibility with other implementations must be ascertained due to differences in the tools and development platforms used. Other optimization mechanisms, as well as improvements to those presented here, can be explored. Of special interest is improving the way the system can deal with flows of notifications with topics which are syntactically different, but which are logically related (e.g. those with numeric values on some dimension). Presently, the system can only do buffering for these topics based on existing subscriptions at rendezvous nodes, but other mechanisms can be developed to further reduce messaging. Other issues, such as reliability and fault tolerance at the messaging level, can also be further developed.

References

[1] OASIS WSN Technical Committee: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn.

[2] Globus Alliance OGSA webpage: http://www.globus.org/ogsa/.

[3] Apache Pubscribe project home: http://ws.apache.org/pubscribe/.

[4] WSRF.NET Project homepage: http://www.cs.virginia.edu/ gsw2c/wsrf.net.html.

[5] N. Jiang, C. Schmidt, V. Matossian, and M. Parashar, "Enabling applications in sensor-based pervasive environments," in *Basenets 2004*, San Jose, CA, October 2004.

[6] S. Graham, D. Hull, and B. Murray, *Web Services Base Notification 1.3 (WS-BaseNotification)*, public review draft 01 ed., Oasis, July 2005. [Online]. Available: http://docs.oasis-open.org/wsn/wsn-ws_base_notification-1.3-spec-pr-02.pdf

[7] D. Chappell and L. Liu, *Web Services Brokered Notification 1.3 (WS-BrokeredNotification)*, public review draft 01 ed., Oasis, July 2005. [Online]. Available: http://docs.oasis-open.org/wsn/wsn-ws_brokered_notification-1.3-spec-pr-02.pdf

[8] W. Vambenepe, *Web Services Topics 1.3 (WS-Topics)*, public review draft 01 ed., Oasis, December 2005. [Online]. Available: http://docs.oasis-open.org/wsn/wsn-ws_topics-1.3-spec-pr-01.pdf

[9] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys*, vol. 35, no. 2, p. 114131, 2003.

[10] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana, "Internet indirection infrastructure," in *Proceedings of ACM SIGCOMM*, Pittsburgh, PA, 2002, pp. 73–86.

[11] C. Schmidt and M. Parashar, "Flexible information discovery in decentralized distributed systems," in *12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12'03)*, 2003.

[12] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of ACM SIGCOMM*, San Diego, CA, August 2001, pp. 149–160.

[13] http://www.planet-lab.org/. [Online]. Available: http://www.planet-lab.org/

[14] N. Jiang, C. Schmidt, and M. Parashar, "A decentralized content-based aggregation service for pervasive environments," June 2006, accepted, International Conference of Pervasive Services (ICPS).

[15] pyGridWare project homepage: http://dsd.lbl.gov/gtg/projects/pyGridWare/.

[16] GT4 tutorial: http://gdp.globus.org/gt4-tutorial/multiplehtml/index.html.

[17] Y. Huang, A. Slominski, C. Herath, and D. Gannon, "Ws-messenger: A web services-based messaging system for service-oriented grid computing," in *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid06)*, 2006.

[18] Eventing specification: http://www.w3.org/Submission/WS-Eventing/.

[19] M. Humphrey, G. Wasson, J. Gawor, J. Bester, S. Lang, I. Foster, S. Pickles, M. McKeown, K. Jackson, J. Boverhof, M. Rodriguez, and S. Meder, "State and events for web services: A comparison of five ws-resource framework and ws-notification implementations," in *14th IEEE International Symposium on High Performance Distributed Computing (HPDC-14)*, Research Triangle Park, NC, July 2005, pp. 24–27.

[20] S. Pallickara and G. Fox, "Naradabrokering: A middleware framework and architecture for enabling durable peer-to-peer grids," in *Proceedings of ACM/IFIP/USENIX International Middleware Conference*, 2003, pp. 41–61.

[21] A. Rowstron and P. Druschel, "Pastry: Scalable, descentralized object location and routing for large-scale peer-to-peer systems," in *Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, November 2001, pp. 329–350.

[22] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proceedings of ACM SIGCOMM*, San Diego, CA, 2001, pp. 161–172.

[23] R. Baldoni, C. Marchetti, A. Virgillito, and R. Vitenberg, "Content-based publish-subscribe over structured overlay networks," in *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS '05)*, Columbus, OH, June 2005.

[24] D. Tam, R. Azimi, and H.-A. Jacobsen, "Building content-based publish/subscribe systems with distributed hash tables," *Lecture Notes in Computer Science*, vol. 2944, pp. 138–152, 2004.

[25] I. Aekaterinidis and P. Triantafillou, "Internet scale string attribute publish/subscribe data networks," in *Proceedings of the ACM 14th Conference on Information and Knowledge Management (CIKM)*, Bremen, Germany, October 2005.

[26] A. Gupta, O. D. Sahin, D. Agrawal, and A. E. Abbadi, "Meghdoot: Content-based publish/subscribe over p2p networks," *Lecture Notes in Computer Science*, vol. 3231, pp. 254–273, 2004.